Technical Report

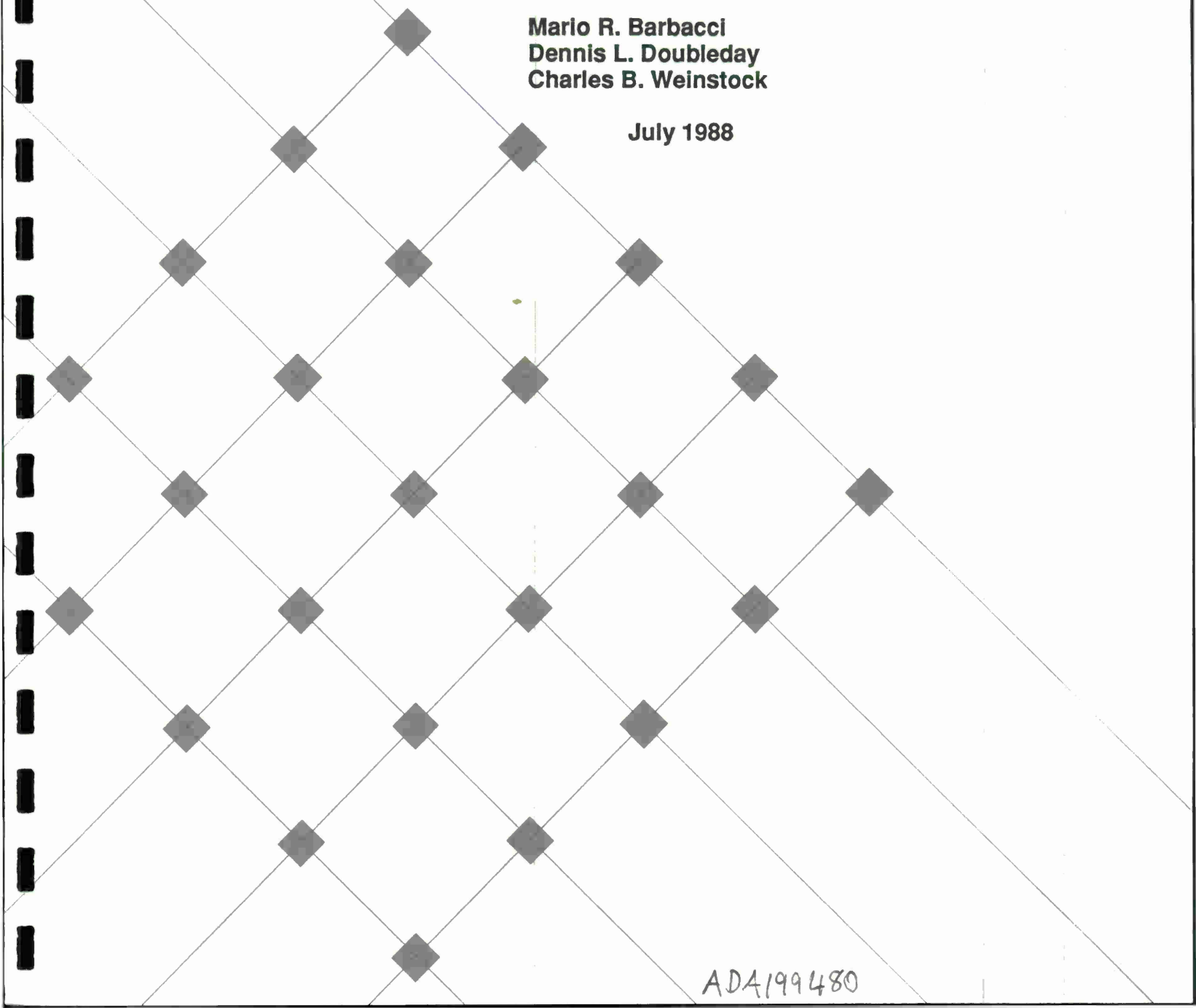
CMU/SEI-88-TR-18
ESD-TR-88-019

Carnegie-Mellon University
Software Engineering Institute

The Durra Runtime Environment

Mario R. Barbacci
Dennis L. Doubleday
Charles B. Weinstock

July 1988



ADA199480

Technical Report
CMU/SEI-88-TR-18
ESD-TR-88-19
July 1988

The Durra Runtime Environment

Mario R. Barbacci
Dennis L. Doubleday
Charles B. Weinstock

Software for Heterogeneous Machines Project

Approved for public release.
Distribution unlimited.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESD/XRS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Karl Shingler
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Copyright © 1988 Carnegie Mellon University

This document is available through the Defense Technical Information Center. DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145.

Copies of this document are also available through the National Technical Information Services. For information on ordering, please contact NTIS directly: National Technical Information Services, U.S. Department of Commerce, Springfield, VA 22161.

Use of any trademarks in this handbook is not intended in any way to infringe on the rights of the trademark holder.

Table of Contents

1. Introduction to Durra 1
- 1.1. Type Declarations 3
- 1.2. Task Descriptions 4
  - 1.2.1. Interface Information 4
  - 1.2.2. Attribute Information 5
  - 1.2.3. Behavioral Information 5
  - 1.2.4. Structural Information 5
- 1.3. Task Selections 7
- 1.4. Runtime Components 7

2. The Scheduler 9
- 2.1. Scheduler Instructions 9
- 2.2. Execution 10

3. The Server 13

4. Application Tasks 15
- 4.1. Remote Procedure Calls 15

5. A Durra Example 17

6. Compilation and Execution Commands 23

7. Implementation Notes 27

References 29

Appendix A. The C Interface Specification 31

Appendix B. The Ada Interface Specification 33

The Durra Runtime Environment

Abstract. Durra is a language designed to support PMS-level programming. PMS stands for Processor-Memory-Switch, the name of the highest level in the hierarchy of digital systems. An application or PMS-level program is written in Durra as a set of *task descriptions* and *type declarations* that prescribes a way to manage the resources of a heterogeneous machine network. The application describes the tasks to be instantiated and executed as concurrent processes, the types of data to be exchanged by the processes, and the intermediate queues required to store the data as they move from producer to consumer processes.

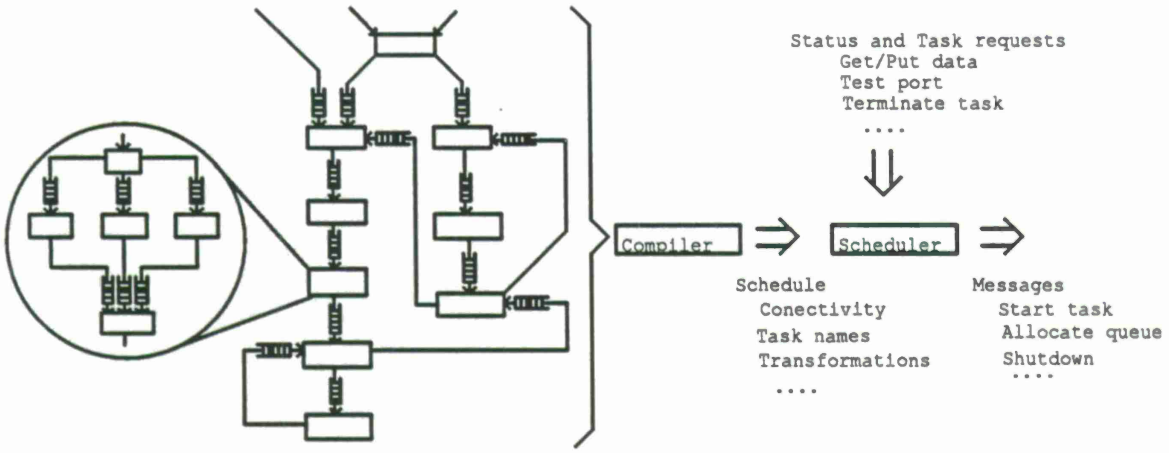
This report describes the Durra Runtime Environment. The environment consists of three active components: the application tasks, the Durra server, and the Durra scheduler. After compiling the type declarations, the component task descriptions, and the application description, the application can be executed by starting an instance of the server on each processor, starting an instance of the scheduler on one of the processors, and downloading the component *task implementations* (i.e., the programs) to the processors. The scheduler receives as an argument the name of the file containing the scheduler program generated by the compilation of the application description. This step initiates the execution of the application.

1. Introduction to Durra

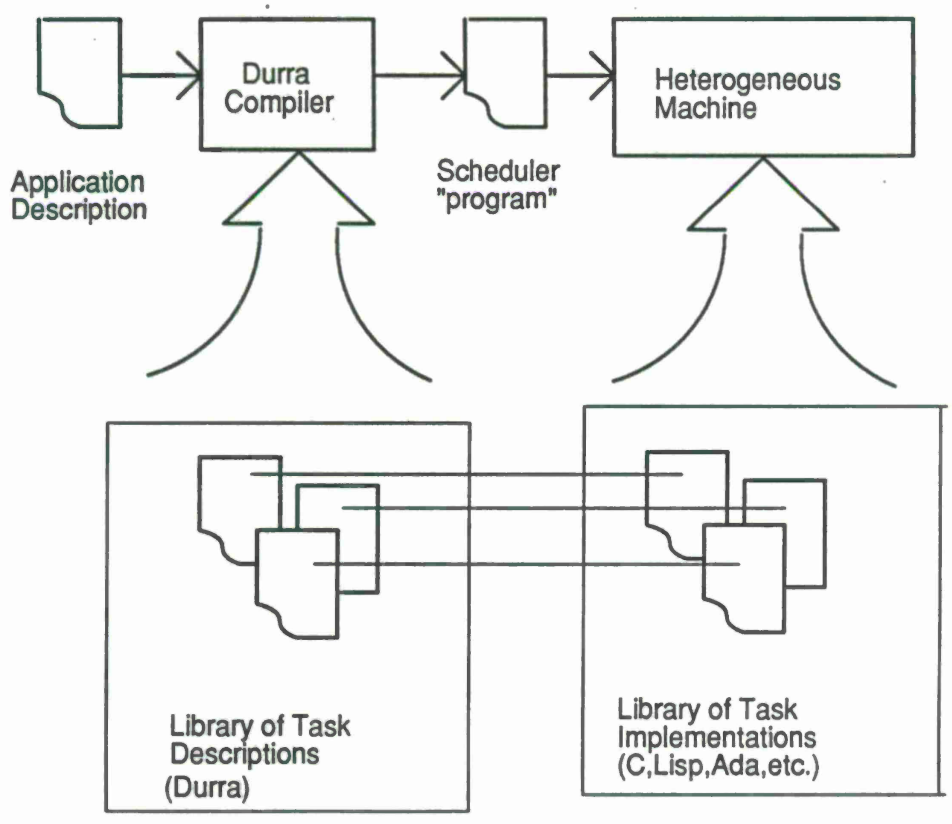
Durra [1, 2] is a language designed to support PMS-level programming. PMS stands for Processor-Memory-Switch, the name of the highest level in the hierarchy of digital systems introduced by Bell and Newell in [3]. An application or PMS-level program is written in Durra as a set of *task descriptions* and *type declarations* that prescribes a way to manage the resources of a heterogeneous machine network. The application describes the tasks to be instantiated and executed as concurrent processes, the types of data to be exchanged by the processes, and the intermediate queues required to store the data as they move from producer to consumer processes.

Because tasks are the primary building blocks, we refer to Durra as a *task-level description language*. We use the term "description language" rather than "programming language" to emphasize that a Durra application is not translated into object code in some kind of executable (conventional) "machine language" (the domain of the Instruction Set Processor or ISP level introduced in [3]). Instead, a Durra application is a description of the structure and behavior of a logical machine to be synthesized into resource allocation and scheduling directives, which are then interpreted by a combination of software, firmware, and hardware in each of the processors and buffers of a heterogeneous machine (the domain of PMS). This is the translation process depicted in Figure 1-1.a.

We see three distinct phases in the process of developing an application using Durra: the creation of a library of tasks, the creation of an application using library tasks, and the execution of the application. These three phases are illustrated in Figure 1-1.b.



a -- Compilation of a PMS-Level Program Graph



b -- Developing a Durra Application

Figure 1-1: Scenario

During the first phase, the developer of the application writes descriptions of the data types (image buffers, map database queries, etc.) and of the tasks (sensor processing, feature recognition, map database management, etc.).

Type declarations are used to specify the format and properties of the data that will be produced and consumed by the tasks in the application. As we will see later in this chapter, tasks communicate through typed ports; and for each data type in the application, a type declaration must be written in Durra, compiled, and entered in the library.

Task descriptions are used to specify the properties of a task implementation (a program). For a given task, there may be many implementations, differing in programming language (e.g., C or assembly language), processor type (e.g., Motorola 68020 or IBM 1401), performance characteristics, or other attributes. As in the case of type declaration, for each implementation of a task, a task description must be written in Durra, compiled, and entered in the library. A task description includes specifications of a task implementation's performance and functionality, the types of data it produces or consumes, the ports it uses to communicate with other tasks, and other miscellaneous attributes of the implementation.

During the second phase, the user writes an *application description*. Syntactically, an application description is a single task description and could be stored in the library as a new task. This allows writing of hierarchical application descriptions. When the application description is compiled, the compiler generates a set of resource allocation and scheduling commands or instructions to be interpreted by the scheduler.

During the last phase, the scheduler loads the task implementations (i.e., programs corresponding to the component tasks) into the processors and issues the appropriate commands to execute the programs.

1.1. Type Declarations

The data types transmitted between the tasks are declared independently of the tasks. In Durra, these data type declarations specify scalars (of possible variable length), arrays, simple record types, or unions of other types, as shown in the following examples:

```
type packet is size 128 to 1024;
                                        -- Packets are of variable length.
type tail is array (5 10) of packet;
                                    -- Tails are 5 by 10 arrays of packets.
type rec is record (rows: integer, columns: integer, data: packet);
                                -- Recs consist of two integers and a packet.
type mix is union (head, tail);
                                        -- Mixs could be heads or tails.
```

1.2. Task Descriptions

Task descriptions are the building blocks for applications. Task descriptions include the following information (Figure 1-2): (1) its interface to other tasks (**ports**) and to the scheduler (**signals**); (2) its **attributes**; (3) its functional and timing **behavior**; and (4) its internal **structure**, thereby allowing for hierarchical task descriptions.

```
task task-name
  ports              -- Used for communication between a process and a queue
    port-declarations

  signals   -- Used for communication between a process and the scheduler
    signal-declarations

  attributes         -- Used to specify miscellaneous properties of the task
    attribute-value-pairs

  behavior -- Used to specify functional and timing behavior of the task
    requires predicate
    ensures predicate
    timing timing expression
  structure          -- A graph describing the internal structure of the task
    process-declarations     -- Declaration of instances of internal subtasks

    bind-declarations        -- Mapping of internal ports to this task's ports

    queue-declarations -- Means of communication between internal processes.

    reconfiguration-statements          -- Dynamic modifications to the structure
end task-name
```

Figure 1-2: A Template for Task Descriptions

1.2.1. Interface Information

The interface information defines the ports of the processes instantiated from the task and the signals used by these processes to communicate with the scheduler:

```
ports
    in1: in heads;
    out1, out2: out tails;
signals
    stop, start, resume: in;
    range_error, format_error: out;
```

A port declaration specifies the direction and type of data moving through the port. An **in** port takes input data from a queue; an **out** port deposits data into a queue. A signal declaration specifies only the direction of the scheduler messages. An **in** signal is a message that a process can receive from the scheduler; an **out** signal is a message that a process can send to the scheduler; an **in out** signal is used for both directions of communication.

1.2.2. Attribute Information

The attribute information specifies miscellaneous properties of a task. Attributes are a means of indicating pragmas or hints to the compiler and/or scheduler. In a task description, the developer of the task lists the actual value of a property; in a task selection (Section 1.3), the user of a task lists the desired value of the property. Example attributes include author, version number, programming language, file name, and processor type:

```
attributes
    author = "jmw";
    implementation = "program_name";
    Queue_Size = 25;
```

1.2.3. Behavioral Information

The behavioral information specifies functional and timing properties about the task. The functional information part of a task description consists of a pre-condition on what is required to be true of the data coming through the input ports, and a post-condition on what is guaranteed to be true of the data going out through the output ports. The timing expression describes the behavior of the task in terms of the operations it performs on its input and output ports. For additional information about the syntax and semantics of the functional and timing behavior description, see the Durra reference manual [1].

1.2.4. Structural Information

The structural information defines a process-queue graph (e.g., Figure 1-1.a) and possible dynamic reconfiguration of the graph. Three kinds of declarations and one kind of statement can appear as structural information. This is illustrated in Figure 1-3, which shows the Durra (i.e., textual) version of the example in Figure 1-1.a.

A process declaration of the form

process_name : **task** *task_selection*

creates a process as an instance of the specified task. Since a given task (e.g., convolution) might have a number of different implementations that differ along different dimensions such as algorithm used, code version, performance, processor type, the task selection in a process declaration specifies the desirable features of a suitable implementation. The presence of task selections within task descriptions provides direct linguistic support for hierarchically structured tasks (Section 1.3).

A queue declaration of the form

queue_name [*queue_size*]: *port_name_1* > *data_transformation* > *port_name_2*

creates a queue through which data flow from an output port of a process (port_name_1) into the input port of another process (port_name_2). Data transformations are operations applied to data coming from a source port before they are delivered to a destination port.

A port binding of the form

task_port = *process_port*

```
task ALV
  ports
    in1, in2: in map_database;
    in3: in destination;
  structure
    process
      navigator:            task navigator
                              attributes author = "jmw";
                            end navigator;
      road_predictor:       task road_predictor;
      landmark_predictor:   task landmark_predictor;

      . . . . . .
      ct_process:           task corner_turning;
    queue
      q1: navigator.out1          > > road_predictor.in2;
      q2: navigator.out2          > > landmark_predictor.in1;

      . . . . . .
      q12:position_computation.out2> > landmark_predictor.in2;
    bind
      in1 = road_predictor.in1;
      in2 = navigator.in1;
      in3 = navigator.in2;
end ALV;
```

Figure 1-3: Structural Information

maps a port on an internal process to a port defining the external interface of a compound task.

A reconfiguration statement of the form

```
if condition then
    remove   process-names
    process  process-declarations
    queues   queue-declarations
end if;
```

is a directive to the scheduler. It is used to specify changes in the current structure of the application (i.e., process-queue graph) and the conditions under which these changes take effect. Typically, a number of existing processes and queues are replaced by new processes and queues, which are then connected to the remainder of the original graph. The reconfiguration predicate is a Boolean expression involving time values, queue sizes, and other information available to the scheduler at runtime.

1.3. Task Selections

As described in the previous section, a process is an instance of a task specified in the process declaration. Given that a number of alternative task implementations might exist in the library, it is necessary to specify in the process declaration the desirable properties of the appropriate implementation. Here are some examples of process declarations, which in turn are used to select tasks:

```
process
  p1: task finder;
  p2: task finder ports foo: in heads, bar: out tails; end finder;
  p3: task finder attributes author="mrb"; end finder;
```

In a process declaration, an instance of a task is bound to the name of the process. The task selection contains at least the name of a task and (optionally) interface, attribute, and behavior requirements (i.e., anything but structural information) and is used to select among a number of alternative task implementations.

A task can therefore be identified and selected from the library just by its name (if the name is unique in the library), by its interface properties (e.g., port types), by its attributes (e.g., version number), by its functional or timing behavior (e.g., a pre-condition), or by any combination of these.

1.4. Runtime Components

There are three active components in the Durra runtime environment: the application tasks, the Durra server, and the Durra scheduler. Figure 1-4 shows the relationship among these components.

After compiling the type declarations, the component task descriptions, and the application description, as described previously and illustrated in Figure 1-1, the application can be executed by performing the following operations:

1. The component task implementations (Chapter 4) must be stored in a special directory in the appropriate processors. The directory name (Chapter 7) is known to the Durra servers and scheduler.
2. An instance of the Durra server (Chapter 3) must be started in each processor.
3. The scheduler (Chapter 2) must be started in one of the processors. The scheduler receives as an argument the name of the file containing the scheduler program generated by the compilation of the application description. This step initiates the execution of the application.

In the remainder of this report, we describe in detail the three components of the runtime environment: the scheduler, the server, and the application task.

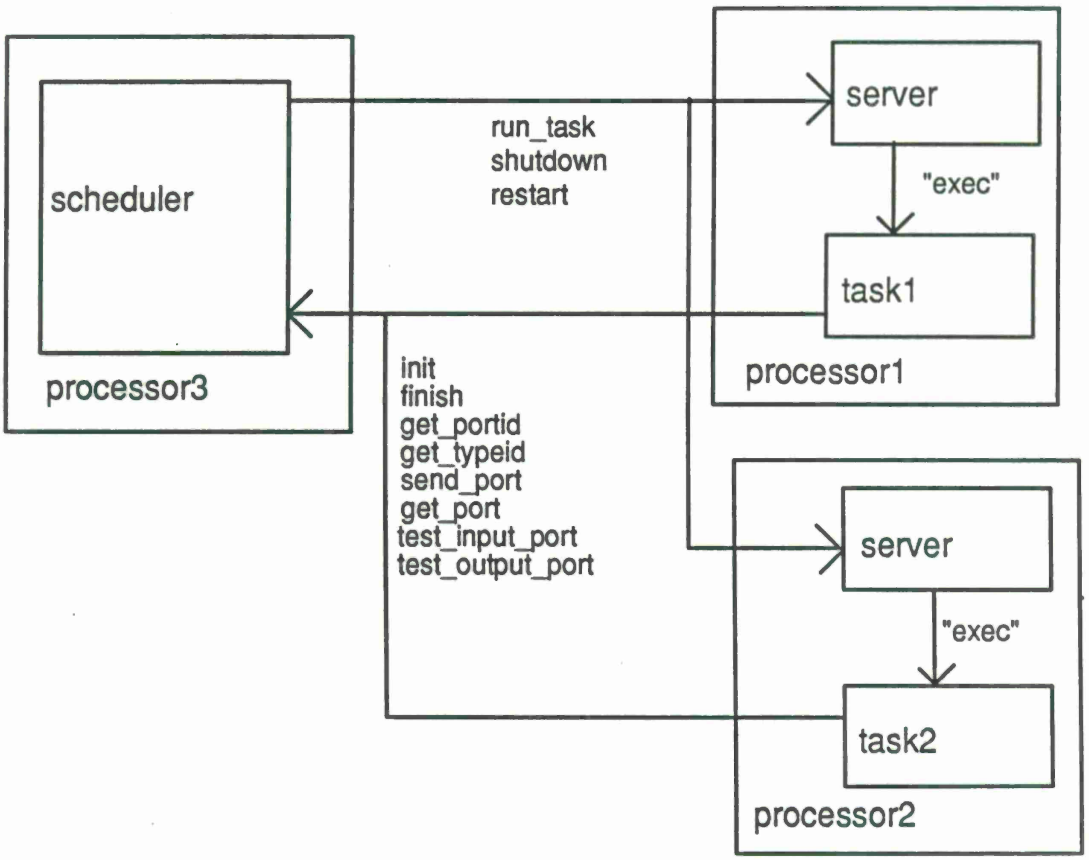


Figure 1-4: The Durra Runtime Environment

2. The Scheduler

The scheduler is the part of the Durra runtime system responsible for starting the tasks, establishing communication links, and monitoring the execution of the application. In addition, the scheduler implements the predefined tasks (**broadcast**, **merge**, and **deal**) and the data transformations described in [1]. The scheduler is invoked with a UNIX command of the form:

sched *scheduler_program*

where the argument is the name of the file containing the scheduler instructions generated by the Durra compiler (See Chapter 6 for additional information about this command.)

2.1. Scheduler Instructions

The scheduler currently understands the following instructions produced by the Durra compiler:

port_allocate This instruction takes four parameters: 1) the name of the task that is defining the port; 2) the name of the port; 3) the type of the port; and 4) the direction of the port (i.e., whether it is an input or output port). The naming convention NAME1.NAME2 is used to represent process NAME2 within task NAME1.

```
(port_allocate MAIN.PA OUT1 STRING out)
```

queue_allocate This instruction takes seven parameters: 1) the name of the task defining the queue; 2) the name of the queue; 3) the name of the task writing to the queue; 4) the name of the port that task will use; 5) the name of the task reading from the queue; 6) the name of the port that task will use; and 7) the maximum number of elements that the queue can hold (if the queue declaration does specify a size, this parameter is 0). If transformations are specified in the queue definition, they appear as separate instructions.

```
(queue_allocate MAIN QAM PA OUT1 PM IN1 0 0 STRING)
```

transformation This instruction takes three parameters: 1) the name of the task defining the queue; 2) the name of the queue; and 3) the list of transformations to be applied to the elements in the queue.

task_load This instruction takes three parameters: 1) the task name; 2) a processor to run it on (either a specific processor or a class of processors); and 3) the name of the executable file.

```
(task_load MAIN.PC VAX  "sink_task2")
```

buffer_task This instruction takes three parameters: 1) the task name; 2) the kind of buffer task (currently **broadcast**, **merge**, and **deal** are implemented); and 3) the "mode" of the buffer task (e.g., fifo, round-robin).

```
(buffer_task MAIN.PM MERGE FIFO)
```

equal_port
This instruction takes four parameters: 1) the task name of the main task; 2) the port name used by the main task; 3) the task name of the subtask; and 4) the port name used by the subtask. This instruction implements the port bind declarations.

```
(equal_port MAIN IN PM IN2)
```

type
This instruction takes a variable number of parameters: All have the following two parameters: 1) the name of the type; and 2) the kind of type (ARRAY, SIZE, UNION, RECORD). Additionally, an ARRAY type has the type of elements of the array, followed by zero or more bounds. A SIZE (scalar) type has an upper and lower bound for the length, in bits. A UNION type has a list of the types making up the union. A RECORD type has a list of the field names and types.

```
(type FLOAT SIZE 32 32)
(type FLOAT_IMAGE ARRAY FLOAT)
(type GENERAL_IMAGE UNION FLOAT_IMAGE INTEGER_IMAGE)
(type RECTANGLE RECORD
                        FIRST_ROW:INTEGER LAST_ROW:INTEGER
                      FIRST_COL:INTEGER LAST_COL:INTEGER)
```

source
This instruction takes two parameters, 1) a task name, and 2) the value of the "source" attribute specified in the task description, if any. If the source attribute is not specified in the task description, the parameter is the name of the syntax tree produced by the Durra compiler (See section 6 for file-naming conventions). This information might be used by the task implementation to examine its own Durra description. This information is passed to the indicated application task at startup.

```
(source MAIN.DISPLAY "display.durra.TREE")
```

Section 5 contains a complete example of a Durra application. It consists of two type declarations, three component task descriptions (and their implementations), the application description, and the scheduler instructions produced by the Durra compiler.

2.2. Execution

After the file with instructions has been read and processed, the scheduler is ready to start the execution of the application. In the current UNIX implementation, this is done by performing the following steps:

1. Allocate a UNIX socket for communication with the application tasks. A UNIX socket is a special intertask communications port defined by the UNIX operating system.
2. Establish communication with each of the processors running a Durra server (Chapter 3).
3. For each of the **task_load** instructions, issue to the appropriate server a **run_task** remote procedure call (Chapter 3).
4. Listen in on the UNIX socket allocated in the first step for remote procedure calls from the application tasks (Chapter 4.1).

5. Process the remote procedure calls from the application tasks (Section 4.1).

The scheduler waits until all tasks have completed their execution before it, in turn, finishes its execution.

3. The Server

The server is responsible for starting tasks on its corresponding processor, as directed by the scheduler. One instance of the server must be running on each processor that is to (potentially) execute Durra tasks.

When a server begins execution, it listens in on a predetermined socket for messages from the scheduler. Once a communication channel is open, the scheduler communicates with the server using the following set of remote procedure calls:

run_task(name, host, port, id, source_attribute, debug_level, task_directory)
: Requests the server to run (as an independent process) the executable file "name," a task implementation. See Chapter 7 for details about locations and directories for task implementations.

  The parameters are: the host and socket where the scheduler will be listening for messages from the task; the identifier (an integer) that the task should use in identifying itself; the value of the "source" attribute (used in the scheduler **source** instruction, described in section 2.1); the level of information logged for debugging purposes; and finally, the directory where the server can find the task implementation and where tracing files, if any, will be created (Chapter 7).

shutdown(exit_code)
: Requests the server to terminate and exit.

restart()
: Reinitialize the server.

The server sits in a loop responding to the above requests from the scheduler, executing them as directed.

4. Application Tasks

The component task implementations making up a Durra application can be written in any language for which a Durra interface has been provided. As of this writing, there are Durra interfaces for both C and Ada. The C interface appears as Appendix A. The Ada interface appears as Appendix B.

When a task is started, the scheduler supplies it with the following information (via a server): the name of the host on which the scheduler is executing, the UNIX socket on which the scheduler is listening for communications from the task, a small integer to be used in identifying the task, and an application specific string as specified in the "source" attribute in the task description (used by the scheduler **source** instruction, described in section 2.1). The first three parameters are necessary to establish proper communication with the scheduler. The source parameter is provided for the convenience of the task implementation. These parameters are provided to the task by the server, which in turn obtains them, via the **run_task** instruction, from the scheduler (See Chapter 3).

4.1. Remote Procedure Calls

Application tasks use the interface to communicate with other tasks. From the point of view of the task implementation, this communication is accomplished via procedure calls, which return only when the operation is completed. The following remote procedure calls (RPCs) are provided:

Init() Opens a connection to the scheduler. There are several hidden parameters supplied to the interface module by the server, which in turn obtains them, via the **run_task** instruction, from the scheduler (Chapter 3).

finish() Informs the scheduler that the task is terminating.

get_portid(**in** name; **out** portid, bound, size)
Given a port name, returns a small integer port identifier to be used in referring to that port. The name of the port must correspond to one of the ports used in the task description. This call also returns the number of elements that can be stored in the queue associated with the port ("bound") and the size of the elements ("size"). If the size is variable, "size" is set to zero.

get_typeid(**in** name; **out** typeid, size)
Given a type name, returns a small integer type identifier to be used in referring to that type. The name of the type must correspond to one of the type declarations stored in the library. This call also returns the size of elements of that type. If the size is variable (e.g., variable length strings, or **union** types of different sized types), "size" is set to zero.

send_port(**in** portid, data_address, count, typeid)
Sends a block of "count" bytes of data of type "typeid," stored at "data_address," to port "portid."

get_port(**In** portid, data_address; **out** count, typeid)
Receives "count" bytes of data of type "typeid," from port "portid," stored at "data_address."

test_input_port(**In** portid; **out** count, type_of_next, size_of_next)
If "portid" is associated with an input port, **test_input_port** returns the number of elements in the queue connected to the port and, if that number is at least one, returns the type and size of the next element in the queue as well. Thus, **test_input_port** returns zero in "count" if a **get_port** call will block, and a positive number if it will succeed.

The additional output parameters "type_of_next" and "size_of_next" allow the calling task to allocate an appropriate buffer area before making a call to **get_port**. If "portid" is associated with an output port, **test_input_port** records an error message and returns zero in each **out** parameter.

test_output_port(**In** portid; **out** count)
If "portid" is associated with an output port, **test_output_port** returns the number of available slots in the queue connected to the port. Thus, **test_output_port** returns zero if a **send_port** call will block, and a positive number if it will succeed. If "portid" is associated with an input port, **test_output_port** records an error message and returns zero.

Using this collection of scheduler calls, Durra tasks typically would exhibit the following behavior:

1. Call the **init** function to establish communication with the scheduler.
2. Call **get_portid** for each of the task ports (these ports must correspond to the ports used in the task description).
3. Call **get_typeid** for each of the task types (these types must correspond to the data types used in the task description).
4. Call **send_port** and **get_port** as necessary to send and receive data.
5. Call **finish** to break communication with the scheduler.

This behavior is illustrated by an example application in the next chapter.

5. A Durra Example

This chapter contains a complete example of a Durra application. It consists of two type declarations, three component task descriptions (and their implementations), the application description, and the scheduler instructions produced by the Durra compiler.

The example (Figures 5-1 and 5-2) illustrates the use of a predefined task, **broadcast**, which is implemented directly by the scheduler. In this application, one task ("taska") is sending out strings of data, and the **broadcast** buffer task is sending it on to two other tasks ("taskb" and "taskc"). The application description ("main") cements all of the component tasks together.

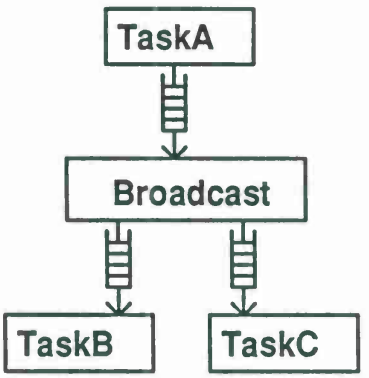


Figure 5-1: Application Structure

"Byte" is the basic type (a scalar type 8 bits long). "String" is an unbounded sequence of bytes.

"Taska" has a single output port, "out1," which produces strings. It can run on any VAX processor and is implemented by the program "source_task." "Taskb" and "taskc" both have a single input port, "in1," which consume strings. These two tasks are implemented by the program "sink_task."

Task "main" is the application description. It specifies the three tasks that make up the application, plus an instance of the predefined task **broadcast**. The structure part specifies the interconnection of those four tasks.

After all of the above files are compiled, the Durra compiler generates a file with instructions to the scheduler. See Section 2.1 for further information about the scheduler instructions. See Chapter 6 for a description of the UNIX commands that invoke the compiler and scheduler-instruction generator. For this example application, the compiler produces the scheduler instructions shown in Figure 5-3.

```
type byte is size 8;

type string is array of byte;
```

a -- Type Declarations

```
task taska
ports
        out1: out string;
attributes
        processor = vax;
        implementation = "source_task";
end taska;

task taskb
ports
        in1: in string;
attributes
        processor = vax;
        implementation = "sink_task";
end taskb;

task taskc
ports
        in1: in string;
attributes
        processor = vax;
        implementation = "sink_task";
end taskc;
```

b -- Component Task Descriptions

```
task main
structure
        process p1: task taska;
                p2: task taskb;
                p3: task taskc;
                pb: task broadcast
                        ports   in1: in string;
                                out1, out2: out string;
                        end broadcast;
        queues  q1b: p1.out1 >> pb.in1;
                qb2: pb.out1 >> p2.in1;
                qb3: pb.out2 >> p3.in1;
end main;
```

c -- Application Description

Figure 5-2: Durra Type Declarations and Task Descriptions

```
(buffer_task MAIN.PB BROADCAST *)
(port_allocate MAIN.P1 OUT1 STRING out)
(port_allocate MAIN.P2 IN1 STRING in)
(port_allocate MAIN.P3 IN1 STRING in)
(port_allocate MAIN.PB IN1 STRING in)
(port_allocate MAIN.PB OUT1 STRING out)
(port_allocate MAIN.PB OUT2 STRING out)
(queue_allocate MAIN Q1B P1 OUT1 PB IN1 0)
(queue_allocate MAIN QB2 PB OUT1 P2 IN1 0)
(queue_allocate MAIN QB3 PB OUT2 P3 IN1 0)
(source MAIN "taskmain.durra.TREE")
(source MAIN.P1 "taska.durra.TREE")
(source MAIN.P2 "taskb.durra.TREE")
(source MAIN.P3 "taskc.durra.TREE")
(task_load MAIN.P1 VAX "source_task")
(task_load MAIN.P2 VAX "sink_task")
(task_load MAIN.P3 VAX "sink_task")
(type BYTE SIZE 8 8)
(type STRING ARRAY BYTE)
```

Figure 5-3: Scheduler Instructions

The **buffer_task** instruction indicates which buffer task to use as process "pb." Buffer tasks are those tasks predefined in the language (**broadcast**, **merge**, and **deal**).

The **port_allocate** instructions set up all the ports in the application. Recall that port names are relative to a process and therefore do not have to be unique across the application.

The **queue_allocate** instructions set up all the queues in the application. For instance one of the instructions allocates the queue named "Q1B," taking input from port "OUT2" of process "MAIN_P1," and outputting to port "IN1" of process "MAIN_PB." The queue has the default queue size.

The **source** instructions pass to the servers and the task implementations the name of the syntax tree produced by the Durra compiler.

The **task_load** instructions associate the program to run (the task implementation), the processor class, and the process name.

The **type** instructions define the application data types "BYTE" and "STRING."

Finally, we complete the example by listing the task implementations for "source_task" and "sink_task." These are written in Ada and are shown in Figures 5-4 and 5-5.

```
with System;
with Interface;
procedure source_task is
  ------------------------------------------------------------------------
  --| A source task has one output port, "out1". Its behavior is to loop
  --| sending 100 strings to out1.
  ------------------------------------------------------------------------
  max_message_size: constant integer := 1000;
  message_buffer: string(1..max_message_size) := (others => ' ');
  out1_port_id, out1_bound: positive;
  out1_type_id, out1_type_size: natural;
begin
  Interface.Init;
  Interface.Get_PortID("out1", out1_port_id, out1_bound, out1_type_size);
  Interface.Get_TypeID("string", out1_type_id, out1_type_size);
  for i in 1..100
  loop
    interface.Send_Port(out1_port_id,
                        message_buffer(1)'address,
                        max_message_size,       --| the real thing
                        out1_type_id);
  end loop;
  Interface.Finish;
end source_task;
```

Figure 5-4: Ada Task Implementation of source_task

```
with System;
with Interface;
procedure sink_task is
   ------------------------------------------------------------------------
   --| A sink task has one input port, "in1". Its behavior is to loop
   --| receiving 100 strings from in1.
   ------------------------------------------------------------------------
   max_message_size: constant integer := 10000;
   message_buffer: string(1..max_message_size);
   in1_port_id, in1_bound: positive;
   in1_type_id, in1_type_size: natural;
   actual_message_type_id, actual_message_size: natural;
begin
   Interface.Init;
   Interface.Get_PortID("in1", in1_port_id, in1_bound, in1_type_size);
   Interface.Get_TypeID("string", in1_type_id, in1_type_size);
   for i in 1..100
   loop
      Interface.Get_Port(in1_port_id,
                         message_buffer(1)'address,
                         actual_message_size,
                         actual_message_type_id);
   end loop;
   Interface.Finish;
end sink_task;
```

Figure 5-5: Ada Task Implementation of sink_task

6. Compilation and Execution Commands

Several UNIX commands have been defined to invoke the various programs that implement Durra:

dall *durra_description_file_name*

The **dall** command invokes the Durra compiler to process a type declaration, a task description, or an application description. A lower-level (i.e., component) task or type must be compiled before a higher-level task or type that uses it.

By convention, Durra source file names have extension "durra." The **dall** command will supply the file extension if it is missing. For example, invoking the command with "some_task" compiles file "some_task.durra." The output files (syntax trees stored in the library) will be named *durra_description_file_name*.TREE.

dcode *application_description_file_name*

The **dcode** command generates the scheduler instructions that execute the application. This command must be issued after all the components and the application descriptions have been compiled with the **dall** command. The output file will be named *application_description_file_name*.SCHED.

dserver {**-d***debug_level*}

The **dserver** command starts a Durra server. It must be issued in each processor that is to execute any of the application tasks. The optional parameter **-d***debug_level* is used for debugging purposes.

dsched { *optional-parameters* } *scheduler_program_file_name*

The **dsched** command starts the scheduler. It must be issued in the processor in which the scheduler will run. The only parameter required is the name of a file containing scheduler instructions. By convention, these files have names of the form *application_name*.**durra.SCHED**. The **dsched** command will supply the appropriate file extensions (".durra" and ".SCHED") if they are missing. For example, invoking the command with "some_application", invokes the scheduler with "some_application.durra.SCHED". There are several optional parameters:

-q*queue_size*
Specifies the default size (number of elements) for those queues whose size was not given in the task or application descriptions. If this argument is not specified, the default queue size is 1.

-d*debug_level*
Specifies the level of tracing and error login information to be recorded at runtime. Is used for debugging purposes.

-c*configuration_file*
Specifies the name of a file containing implementation-dependent information (e.g., the names of the available processors on which tasks can execute).

-s*special_subdirectory*
Specifies the name of the subdirectory in which the task implementations reside and in which tracing and error logging information will be stored. See Chapter 7 for additional details.

dlibrary *switches filenames*
The **dlibrary** command implements a modest library management facility:

dlibrary -a *directory_name*
Adds a pointer to another directory to be searched for imported task descriptions or type declarations.

dlibrary -r *directory_name*
Removes a pointer to another directory. Task descriptions and type declarations defined in that directory are no longer accessible.

dlibrary -d *durra_file_name*
Deletes a task description or type declaration from the library (the source files are not disturbed, only the library entry is deleted). Under normal conditions there is no need to delete library entries using this command because the compiler takes care of inserting or deleting task descriptions or type declarations from the library.

dlibrary -c
Creates a new library (or reinitializes an existing library). This command is normally used on a fresh directory, when starting the development of a new application.

The UNIX Make facility can be used to structure the file dependencies and invoke the compilations in the right order. This is illustrated by the UNIX Make file in Figure 6-1. It defines the file dependencies and order of compilation for all the files used in the example in Chapter 5.

```
example: sink_task source_task taskmain.durra.SCHED

sink_task: sink_taskB.a
        a.make -v -C 'ada -O' -f sink_taskB.a
        a.ld sink_task -o sink_task

source_task: source_taskB.a
        a.make -v -C 'ada -O' -f source_taskB.a
        a.ld source_task -o source_task

taskmain.durra.SCHED:   byte.durra.TREE string.durra.TREE \
                taska.durra.TREE taskb.durra.TREE \
                taskc.durra.TREE taskmain.durra.TREE
        dcode taskmain.durra

byte.durra.TREE: byte.durra
        dall byte.durra

string.durra.TREE: string.durra byte.durra.TREE
        dall string.durra

taska.durra.TREE: taska.durra string.durra.TREE
        dall taska.durra

taskb.durra.TREE: taskb.durra string.durra.TREE
        dall taskb.durra

taskc.durra.TREE: taskc.durra string.durra.TREE
        dall taskc.durra

taskmain.durra.TREE:  taskmain.durra string.durra.TREE \
                taska.durra.TREE taskb.durra.TREE \
                taskc.durra.TREE
        dall taskmain.durra
```

Figure 6-1: A Make File

7. Implementation Notes

The following implementation details are likely to change as the project evolves.

Special directory and files. The task implementations must be stored in a place where the scheduler and the servers can find them. Depending on the languages and operating systems, the format of the command(s) required to compile, link, and move the task implementations to the processors in which they will execute can vary. In processors running the UNIX operating system, the task implementations must be stored in directory "/usr/hetsim/bin." The name of the runnable program must be specified as the value of the "implementation" attribute in the corresponding task descriptions. Thus, if the value of the implementation attribute is "foo," the runnable program in a UNIX processor would be the file "/usr/hetsim/bin/foo."

In addition to the task implementations, any "source" parameters such as Durra syntax tree files used by the implementations also need to be located on those processors in the "/usr/hetsim/bin" directory. See the **source** scheduler instruction (Section 2.1) and the **run_task** remote procedure call (Chapter 3) for a description of the "source" parameter.

To let the scheduler know what host processors it can count on to execute an application, a special file, "/usr/hetsim/bin/config_txt," must exist in the processor on which the scheduler will execute. Config.txt is a text file that describes each host in the following format (one host per line):

hostname list_of_processor_attribute_values_for_that_host

For instance:

```
ag.sei.cmu.edu VAX UVAX VAXGROUP1 SEIAG
e.sei.cmu.edu VAX UVAX VAXGROUP2 SEIE
cx.sei.cmu.edu SUN SEICX
sei.cmu.edu VAX VAXGROUP1 SEI
```

The scheduler first determines the set of processors matching the processor attribute of the task description and then chooses the least loaded of them (as measured by a raw count of tasks assigned to that processor). Note that a task can be assigned to run on a specific processor by giving it a unique processor attribute value (e.g., SEIAG).

Shutting down and restarting the servers. After all the tasks have completed, the scheduler sends **shutdown** messages (Chapter 3) to the servers, and they have to be started anew for each new application execution, using the **dserver** command (Chapter 6). Killing the servers is not strictly necessary; in the future the servers could be started at boot time, along with the other system servers, in each of the processors, and kept running continuously.

Communication protocols. The scheduler communicates with the servers through a reserved, preallocated UNIX socket, known to the servers.

The current implementation has a single, centralized, scheduler communicating, via TCP/IP, with the task implementations and the Durra servers on the various processors. All communication flows through this central scheduler. A later implementation will distribute the functionality of the scheduler, sometimes even allowing direct task-to-task communication. For implementation on Nectar, the heterogeneous machine currently being designed by the Department of Computer Science at Carnegie Mellon University, a different protocol will replace TCP/IP.

Missing pieces. The scheduler is by no means complete. In particular, work is still needed in the following areas:

- Signals, data transformations, and dynamic reconfigurations are not yet implemented.
- Additional predefined tasks could be implemented by the scheduler. Exactly which ones are necessary will become apparent as we develop a user base.
- An interactive debugging and monitoring facility is necessary for users to debug Durra applications by sending commands to the scheduler.
- A centralized scheduler is a bottleneck. The scheduler functionality should be distributed over a number of processors.

References

[1] M.R. Barbacci and J.M. Wing.
Durra: A Task-Level Description Language.
Technical Report CMU/SEI-86-TR-3, Software Engineering Institute, Carnegie Mellon University, 1986.
Also Technical Report CMU-CS-86-176, Department of Computer Science, Carnegie Mellon University, December 1986, and NTIS Report No. AD-A178 975.

[2] M.R. Barbacci, C.B. Weinstock, and J.M. Wing.
Programming at the Processor-Memory-Switch Level.
In *Proceedings of the 10th International Conference on Software Engineering.* Singapore, April, 1988.

[3] C.G. Bell and Allen Newell.
Computer Structures: Readings and Examples.
McGraw-Hill Book Company, New York, 1971.

Appendix A: The C Interface Specification

```
/* Header file defining Durra Interface operations */

/* Define short external names for linker */

#define get_portid      getpid
#define get_typeid      gettid
#define get_port        getprt
#define send_port       sndprt
#define test_input_port tstinprt
#define test_output_port tstoutprt

extern void init();
/* Synopsis: (Open a communications channel between the
             Durra Scheduler and an application task.)
       void init(initname,      -- The name of the application task.
                 inithost,      -- The host where the scheduler is running.
                 initport,      -- The address of the socket where the
                                -- Scheduler will be listening for traffic.
                 initidentity,  -- ID by which task is known to Scheduler.
                 initdebug);    -- Debugging level
         char *initname;
         char *inithost;
         char *initport;
         char *initidentity;
         char *initdebug;
*/

extern void finish();
/* Synopsis: (Inform the Scheduler that an application
             task is terminating.)

       -- No Parameters
*/

extern void get_portid();
/* Synopsis: (Return the unique ID, the associated queue size bound,
             and the data size expected for the specified port.)
       void get_portid(name,
                       id,
                       bound,
                       size)
         char *name;
         int *id, *bound, *size;
*/

extern void get_typeid();
/* Synopsis: (Return the Scheduler-assigned ID and the size (in bytes)
             of the specified data type.)
       void get_typeid(name,
                       typeid,
                       size)
         char *name;
         int *typeid, *size;
*/

extern void send_port();
```

```
/* Synopsis: (Send "count" bytes of data to the port with the
             specified ID.  If "ty" is nonzero, then the data
             is of the type with typeid "ty", otherwise the
             type is unspecified.)
       void send_port(id,
                      data,
                      count,
                      ty)
         int id,count,ty;
         char *data;
*/

extern void get_port();
/* Synopsis: (Get "count" bytes of data (or the actual number
             of bytes sent, if it is less) from the port with
             the specified ID.  If "ty" is nonzero, then the data
             is of the type with typeid "ty", otherwise the
             type is unspecified.)
       void get_port(id,
                     data,
                     count,
                     ty)
         int id;
         int *ty,*count;
         char *data;
*/

extern int test_input_port();
/* Synopsis: (Return zero if a get_port from the specified port
             will block; otherwise return the positive number
             indicating the number of free spaces in the associated
             queue.  Also return the type of the next element in
             next_type, and its size in next_size)
       int test_input_port(id,next_type,next_size)
         int id,*next_type,*next_size;
*/

extern int test_output_port();
/* Synopsis: (Return zero if a send_port to the specified port
             will block; otherwise return the positive number
             indicating the number of free spaces in the associated
             queue.)
       int test_output_port(id)
         int id;
*/
```

Appendix B: The Ada Interface Specification

```
with U_Env; use U_Env;
with system; use system;
-------------------------------------------------------------------------
package Interface is
   ----------------------------------------------------------------------
   --| Durra Scheduler Interface (Low Level)
   --|
   --| This package provides the interface to the Durra scheduler for tasks
   --| written in Ada.
   --|
   --| The init_* variables are the parameters passed by the server when a
   --| task is initialized. The server in turn gets them from the scheduler.
   ----------------------------------------------------------------------

   -- REVISION HISTORY
   -- 01/03/88    mrb     Package spec created.
   -- 06/13/88    dd      Test_Port expanded to separate routines for
   --                       input and output ports.

   User_Task_Name        : constant STRING(1..Argv(0).s'length) := Argv(0).s;
   Scheduler_Host        : constant STRING(1..Argv(1).s'length) := Argv(1).s;
   Scheduler_Port        : constant STRING(1..Argv(2).s'length) := Argv(2).s;
   User_Process_ID       : constant STRING(1..Argv(3).s'length) := Argv(3).s;
   Scheduler_Debug_Level : constant STRING(1..Argv(4).s'length) := Argv(4).s;
   User_Source_Parameter : constant STRING(1..Argv(5).s'length) := Argv(5).s;

   procedure Finish;

   procedure Get_Port (Port_ID       : in     POSITIVE;
                       Data          : in     System.Address;
                       Data_Size     :    out NATURAL;
                       Type_ID       :    out NATURAL);

   procedure Get_PortId (Port_Name   : in     STRING;
                         Port_ID     :    out POSITIVE;
                         Queue_Bound :    out POSITIVE;
                         Data_Size   :    out NATURAL);

   procedure Get_TypeId (Type_Name   : in     STRING;
                         Type_ID     :    out NATURAL;
                         Type_Size   :    out NATURAL);

   procedure Init;

   procedure Send_Port (Port_ID   : in POSITIVE;
                        Data      : in System.Address;
                        Data_Size : in NATURAL;
                        Type_ID   : in NATURAL);

   procedure Test_Input_Port  (Port_ID            : in     POSITIVE;
                               Type_of_Next_Input :    out NATURAL;
                               Size_of_Next_Input :    out NATURAL;
                               Inputs_in_Queue    :    out NATURAL);

   procedure Test_Output_Port (Port_ID            : in     POSITIVE;
                               Spaces_Available   :    out NATURAL);

end Interface;
```

UNLIMITED, UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-88-TR-18	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-TR-88-019
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INSTITUTE 6b. OFFICE SYMBOL (If applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE
6c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY PITTSBURGH, PA 15213	7b. ADDRESS (City, State and ZIP Code) ESD/XRS1 HANSCOM AIR FORCE BASE, MA 01731
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JOINT PROGRAM OFFICE 8b. OFFICE SYMBOL (If applicable) SEI JPO	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962885C0003
8c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY SOFTWARE ENGINEERING INSTITUTE JPO PITTSBURGH, PA 15213	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO.: — PROJECT NO.: N/A TASK NO.: N/A WORK UNIT NO.: N/A
11. TITLE (Include Security Classification) THE DURRA RUNTIME ENVIRONMENT	
12. PERSONAL AUTHOR(S) MARIO BARBACCI, DENNIS DOUBLEDAY, CHARLES WEINSTOCK	
13a. TYPE OF REPORT FINAL 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) JULY 1988 15. PAGE COUNT 38
16. SUPPLEMENTARY NOTATION	
17. COSATI CODES FIELD / GROUP / SUB. GR.	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) DISTRIBUTED APPLICATIONS, REAL-TIME APPLICATIONS, HETEROGENOUS COMPUTER NETWORKS, TASK SPECIFICATION LANGUAGES

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

DURRA IS A LANGUAGE DESIGNED TO SUPPORT PMS-LEVEL PROGRAMMING. PMS STANDS FOR PROCESSOR-MEMORY-SWITCH, THE NAME OF THE HIGHEST LEVEL IN THE HIERARCHY OF DIGITAL SYSTEMS. AN APPLICATION OR PMS-LEVEL PROGRAM IS WRITTEN IN DURRA AS A SET OF TASK DESCRIPTIONS AND TYPE DECLARATIONS THAT PRESCRIBES A WAY TO MANAGE THE RESOURCES OF A HETEROGENEOUS MACHINE NETWORK. THE APPLICATION DESCRIBES THE TASKS TO BE INSTANTIATED AND EXECUTED AS CONCURRENT PROCESSES, THE TYPES OF DATA TO BE EXCHANGED BY THE PROCESSES, AND THE INTERMEDIATE QUEUES REQUIRED TO STORE THE DATA AS THEY MOVE FROM PRODUCER TO CONSUMER PROCESSES.

THIS REPROT DESCRIBES THE DURRA RUNTIME ENVIRONMENT. THE ENVIRONMENT CONSISTS OF THREE ACTIVE COMPONENTS: THE APPLICATION TASKS, THE DURRA SERVER, AND THE DURRA SCHEDULER. AFTER COMPILING THE TYPE DECLARATIONS, THE COMPONENT TASK DESCRIPTIONS, AND THE APPLICATION DESCRIPTION, THE APPLICATION CAN BE EXECUTED BY STARTING AN

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☒	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED
22a. NAME OF RESPONSIBLE INDIVIDUAL KARL SHINGLER	22b. TELEPHONE NUMBER (Include Area Code) (412) 268-7630 22c. OFFICE SYMBOL SEI JPO

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

UNLIMITED, UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

BLOCK 19. CONTINUED

INSTANCE OF THE SERVER ON EACH PROCESSOR, STARTING AN INSTANCE OF THE SCHEDULER ON ONE OF THE PROCESSORS, AND DOWNLOADING THE COMPONENT TASK IMPLEMENTATIONS (I.E., THE PROGRAMS) TO THE PROCESSORS. THE SCHEDULER RECEIVES AS AN ARGUMENT THE NAME OF THE FILE CONTAINING THE SCHEDULER PROGRAM GENERATED BY THE COMPILATION OF THE APPLICATION DESCRIPTION. THIS STEP INITIATES THE EXECUTION OF THE APPLICATION.